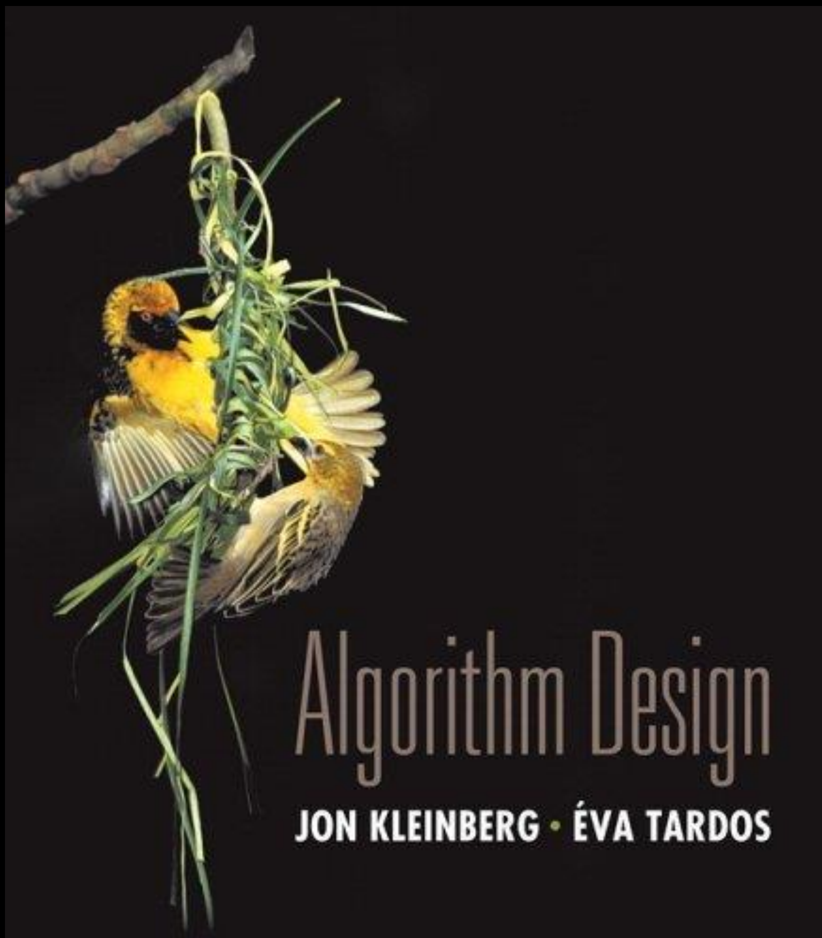


Chapter 6

Dynamic Programming

Slides by Kevin Wayne.

Algorithmic Paradigms

Greed. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of sub-problems with repetitions, and build up solutions to larger and larger sub-problems.

Optimal substructure

Dynamic Programming History

Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, systems, ….

Some famous dynamic programming algorithms.

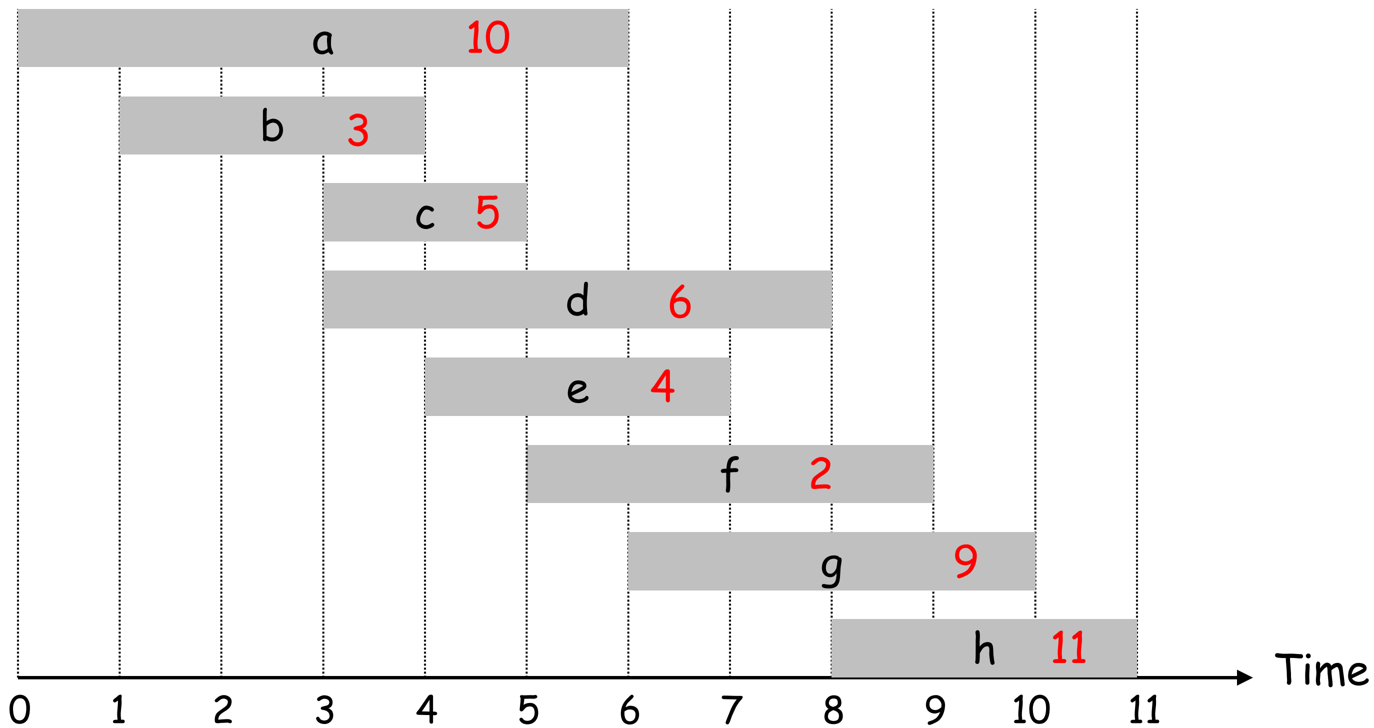
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- DNA sequence comparison.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.

6.1 Weighted Interval Scheduling

Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has value v_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum value subset of mutually compatible jobs.

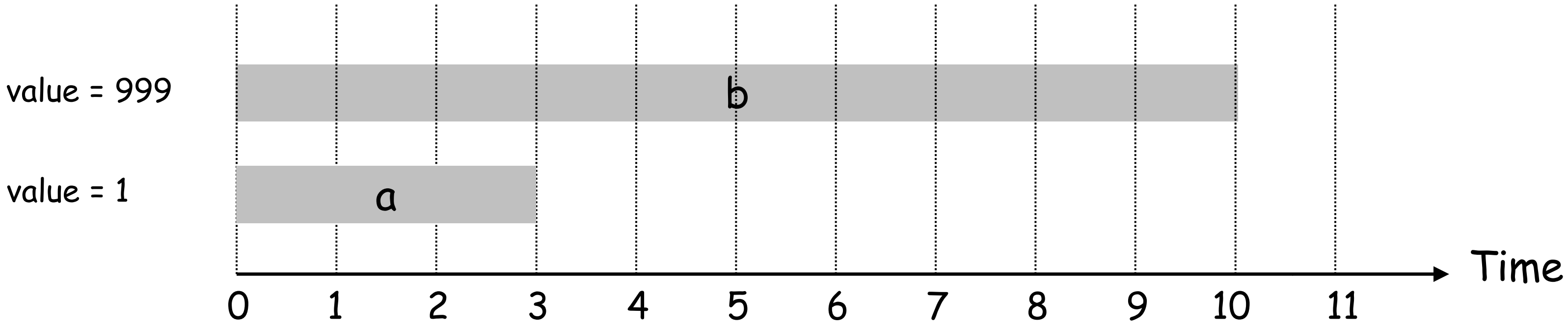


Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all values are 1.

- Consider jobs in ascending order of finish time.
- Add job to solution if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary values are allowed.

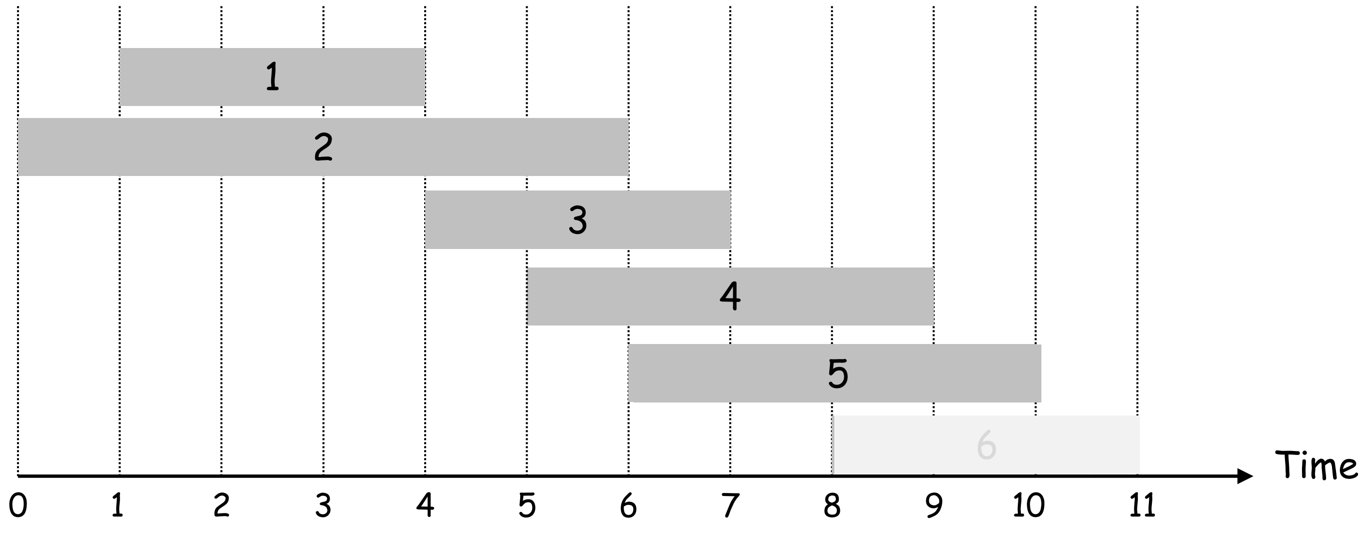


Weighted Interval Scheduling

Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Suppose we want to find optimal solution involving just jobs 1,2,..,5

Need to decide whether to include job 5 or to not include job 5



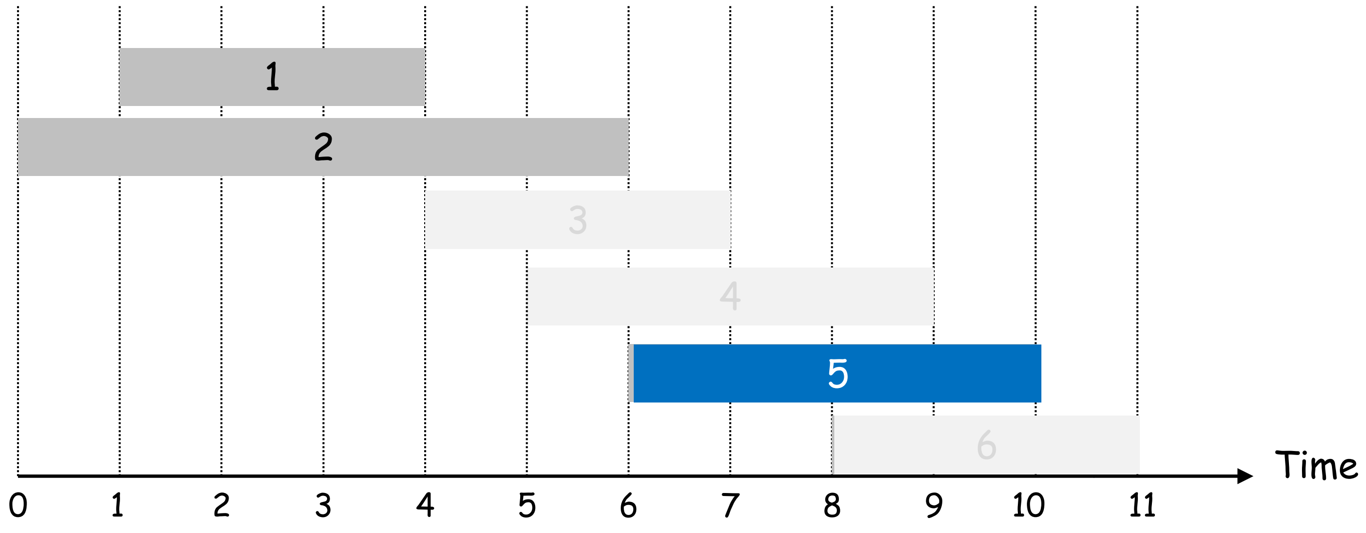
Weighted Interval Scheduling

Label jobs by finishing time: $f_1 \le f_2 \le \ldots \le f_n$.

Suppose we want to find optimal solution involving just jobs 1,2,..,5

Need to decide whether to include job 5 or to not include job 5

1. If include job 5 => also select optimally among jobs 1,2



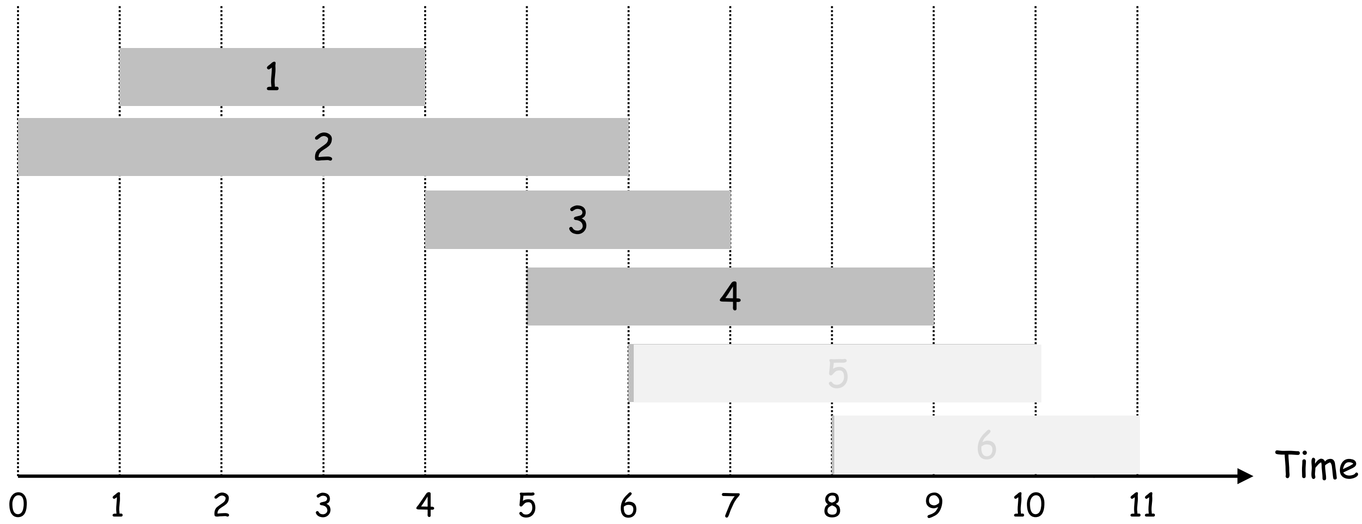
Weighted Interval Scheduling

Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Suppose we want to find optimal solution involving just jobs 1,2,..,5

Need to decide whether to include job 5 or to not include job 5

1. If include job 5 => also select optimally among jobs 1,2
2. If do not include job 5 => select optimally among jobs 1,..,4.

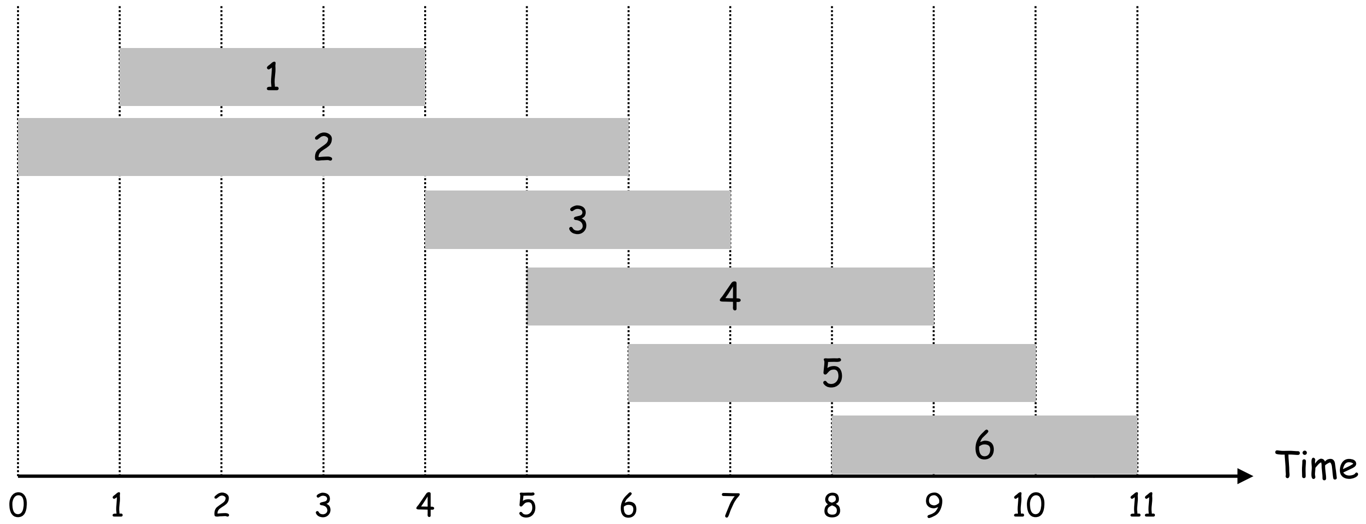


Weighted Interval Scheduling

More generally, define

lastCompat(j) = largest index i < j such that job i is compatible with j.

Ex: lastCompat(5) = 2, lastCompat(4) = 1, lastCompat(1) = 0



Weighted Interval Scheduling

More generally, define

lastCompat(j) = largest index i < j such that job i is compatible with j.

OPT(j) = value of optimal solution to the problem consisting of jobs 1, 2, ..., j.

To compute OPT(j) we have two options:

- Case 1: Solution includes job j.
  - can't use incompatible jobs {lastCompat(j) + 1, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., lastCompat(j) (=OPT(lastCompat(j))
- Case 2: Solution does not include job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1 (=OPT(j-1))

Pick the best option

$$OPT(j) = \begin{cases} 0 & \text{if } \; j = 0 \\ \max\{v_j + OPT(lastCompat(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling

More generally, define

lastCompat(j) = largest index i < j such that job i is compatible with j.

OPT(j) = value of optimal solution to the problem consisting of jobs 1, 2, ..., j.

Optimal substructure

To compute OPT(j) we h

- Case 1: Solution includes job j.
  - can't use incompatible jobs {lastCompat(j) + 1, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., lastCompat(j) (=OPT(lastCompat(j))
- Case 2: Solution does not include job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1 (=OPT(j-1))

Pick the best option

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(lastCompat(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force

$$OPT(j) = \begin{cases} 0 & \text{if } \ j = 0 \\ \max\{v_j + OPT(lastCompat(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

We can use this expression to compute the optimal value OPT(n)

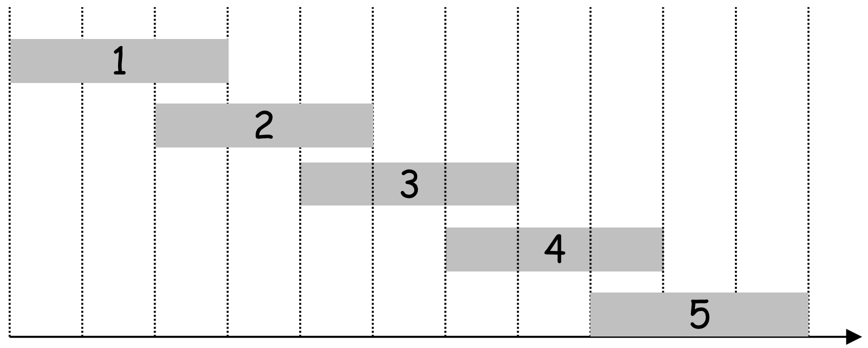
Brute force algorithm.

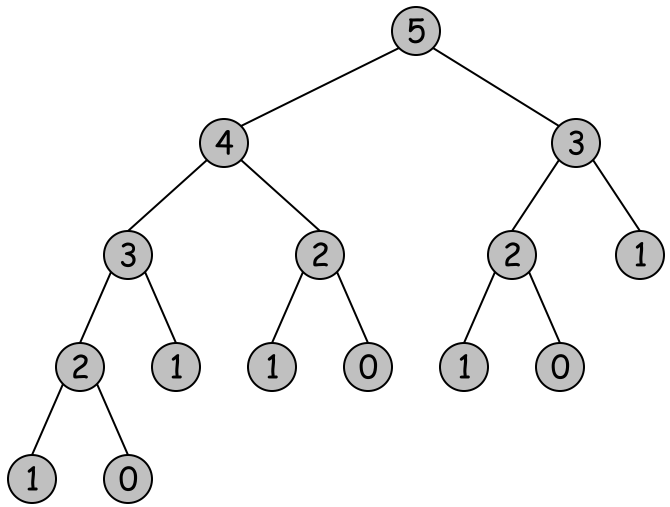
```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute lastCompat(1), lastCompat(2), …, lastCompat(n)

Return Compute-Opt(n)


Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(v_j + Compute-Opt(lastCompat(j)), Compute-Opt(j-1))
}
```

Weighted Interval Scheduling: Brute Force

Observation. This brute force algorithm has takes exponential time because of redundant sub-problems

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.





Q: Any ideas on how to decrease running time?

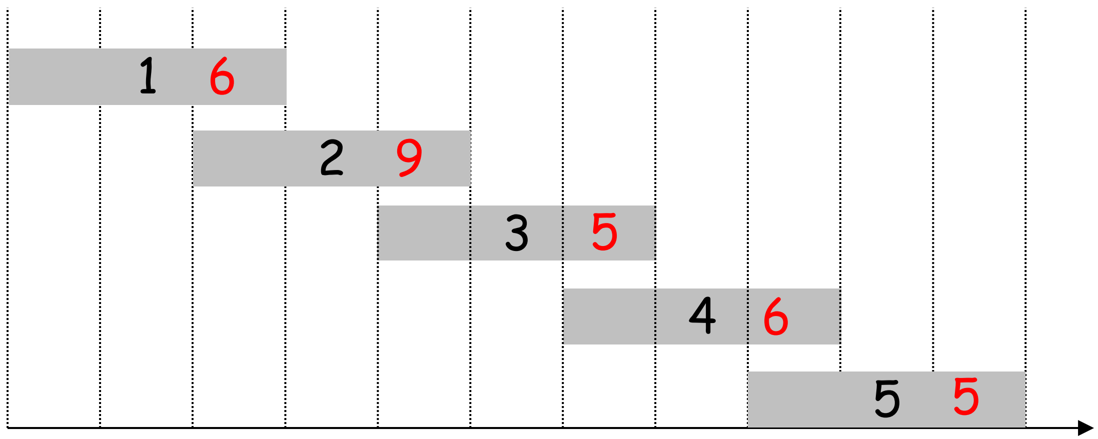
Weighted Interval Scheduling: DP I - Memoization

Dynamic Programming I - Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute lastCompat(1), lastCompat(2), …, lastCompat(n)

M[0] = 0          ← global array, want to have M[j]=OPT(j)
for j = 1 to n
   M[j] = empty
Run M-Compute-Opt(n)

--------
M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(w_j + M-Compute-Opt(lastCompat(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: DP 1 - Memoization

Ex: Run the memoization algorithm on the following instance



M=

Weighted Interval Scheduling: DP 1 - Memoization

- Analysis of running time (we will ignore the time to sort and compute lastCompat)

- Running time = sum of costs of all calls M-Compute-Opt(j), j=0,...,n

- Let us analyze M-Compute-Opt(j) for a fixed j
  - The first time M-Compute-Opt(j) is called, it makes two recursive calls
  - After the first time, it does not make any calls
  - So over the whole execution, M-Compute-Opt(j) makes 2 calls

- Since j=1,...n, the total number of calls is O(n)
- Each call takes constant time
- So the algorithm takes time O(n).

Weighted Interval Scheduling: DP 2 - Bottom-Up

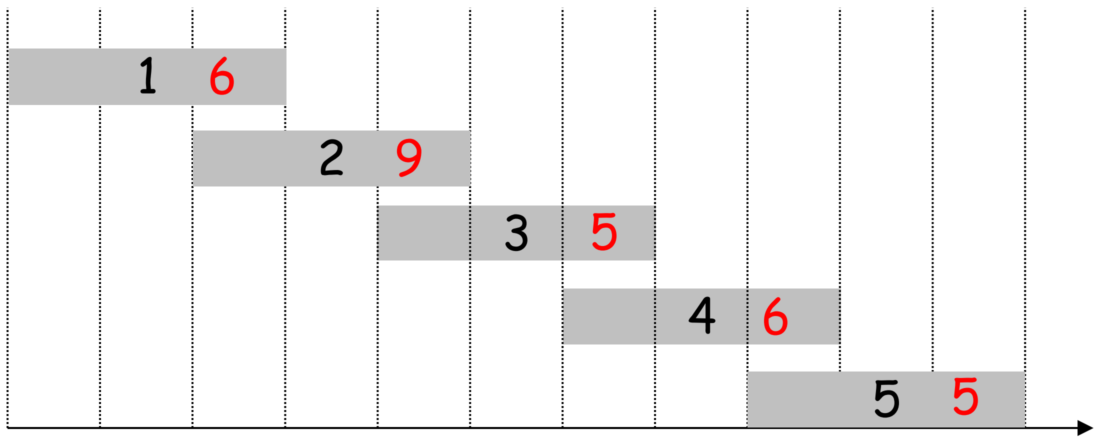
Dynamic Programming 2 - Bottom-up: Fill table M in order M[0], M[1],…

- When we try to fill M[j] we already have all the information needed, namely M[lastCompat(j)] and M[j-1]

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute lastCompat(1), lastCompat(2), …, lastCompat(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[lastCompat(j)], M[j-1])
}
```

Weighted Interval Scheduling: DP 2 - Bottom-Up

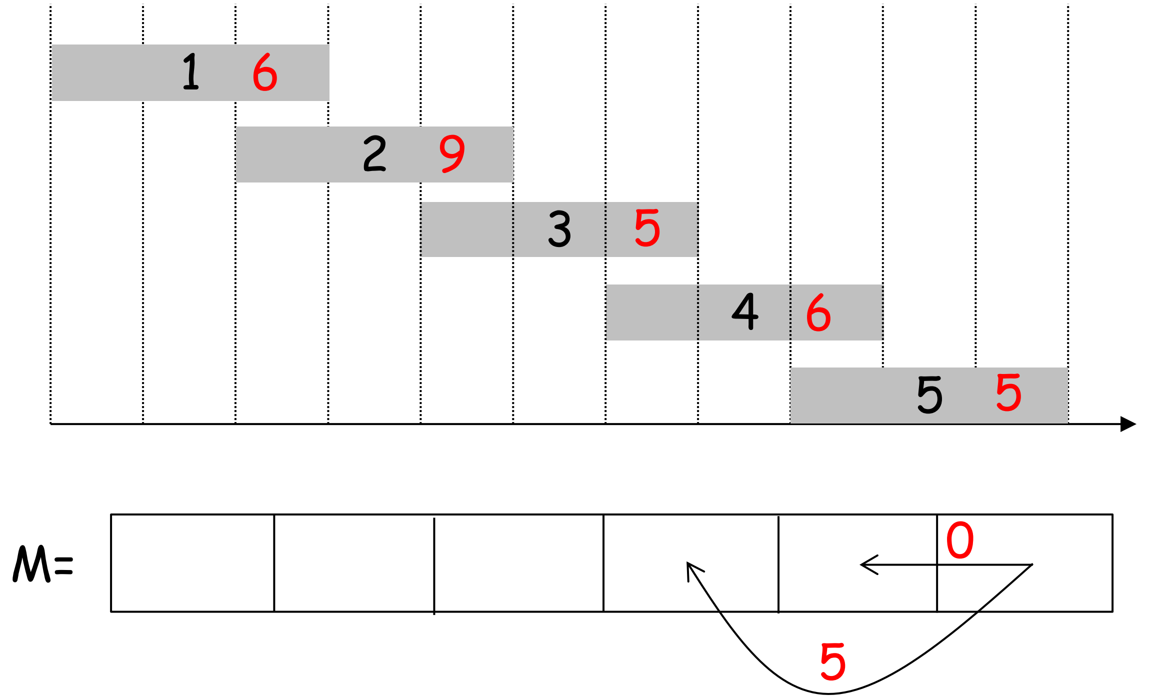
Ex: Run the bottom-up algorithm on the following instance



M=

Weighted Interval Scheduling: DP 3 – Shortest path

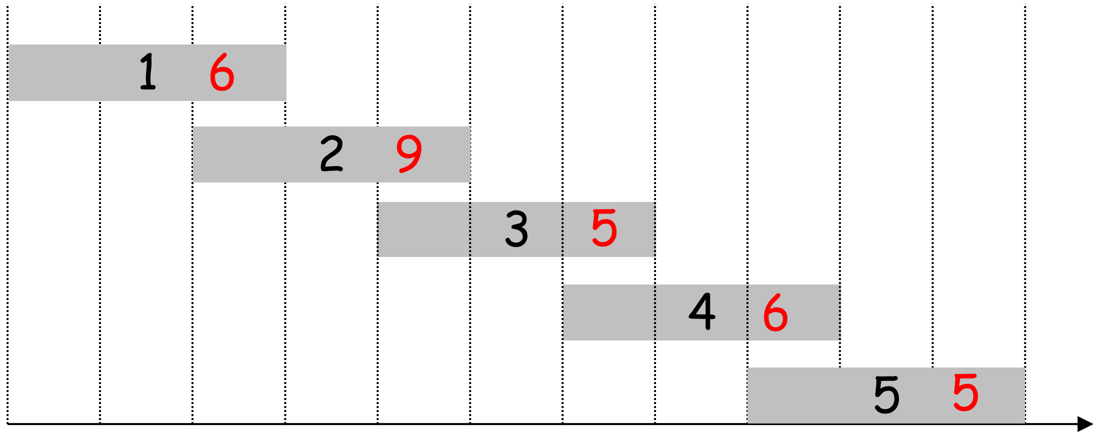
Remark: Every dynamic programming algorithm can also be seen as a shortest/longest path problem

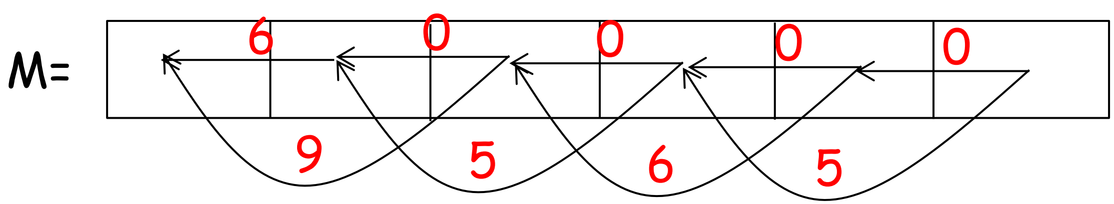


$$OPT(j) = \begin{cases} 0 & \text{if } \ j = 0 \\ \max\{v_j + OPT(lastCompat(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: DP 3 – Shortest path

Remark: Every dynamic programming algorithm can also be seen as a shortest/longest path problem



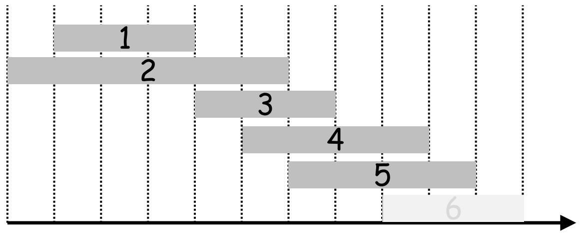


$$OPT(j) = \begin{cases} 0 & \text{if} \quad j = 0 \\ \max\{v_j + OPT(lastCompat(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

How does Dynamic Programming solution looks like?

1) Break problem into sub-problems

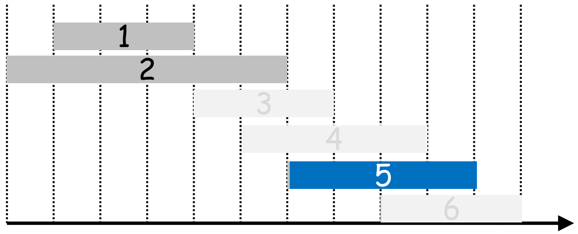
```
Sub-problem i: consider only tasks 1,…,i

OPT(i) = optimal value of sub-prob i
       = best subset of tasks 1,…,i
```

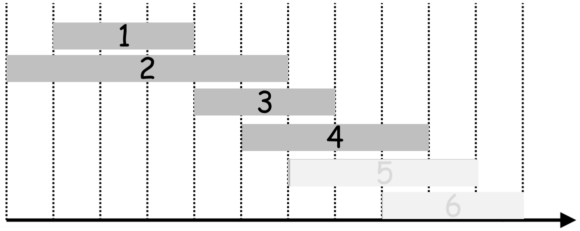


Optimal substructure

2) To solve sub-problem, use smaller sub-problems

Either: `include 5` `does not`






```
OPT(5) = max{ val(5) + OPT(2),     OPT(4)}
```

How does Dynamic Programming solution looks like?

```
OPT(5) = max{ val(5) + OPT(2),    OPT(4)}
```

3) Create table to store optimal value of each sub-problem.
Fill up table in starting from smallest sub-problems
(so always have information needed)

M=						
	1	2	3	4	5	6

Maior subsequência crescente

Maior subsequência crescente

Entrada:

A= (a(1), a(2),...., a(n)) uma sequência de números reais distintos.

Objetivo:

Encontrar a maior subsequência crescente de A

Exemplo

- A= (2, 3, 14, 5, 9, 8, 4)
- (2,3,8) e (3,5,9) são subsequências crescentes de tamanho 3
- As maiores subsequências crescentes de A são 2,3,5,9 e 2,3,5,8

Maior subsequência crescente

Q: Sub-problemas?

- Seja $L(j)$: tamanho da maior subsequência crescente que termina em $a(j)$ ($a(j)$ pertence a subsequência)

- Exemplo $A = (2, 3, 14, 5, 9, 8, 4)$
- $L(1)=1$, $L(2)=2$, $L(3)=3$, $L(4)=3$, $L(5)=4$, $L(6)=4$, $L(7)=3$

- O tamanho da maior subsequência crescente é

$$\max \{ L(1), L(2), \ldots, L(n) \}$$

- Temos a seguinte equação para $L(j)$:
  [Tente para $L(6)$ com $A = (2, 3, 14, 5, 9, 8, 4)$]

$$L(j) = \max_i \{ 1 + L(i) \mid i < j \text{ e } a(i) < a(j) \}, \text{ para } j > 1$$
$$L(1) = 1$$

Maior subsequência crescente

Q: Sub-problemas?

- Seja L(j) : tamanho da maior subsequência crescente que termina em a(j)   (a(j) pertence a subsequência)

- Exemplo A= (2, 3, 14, 5, 9, 8, 4)
- L(1)=1, L(2)=2, L(3)=3, L(4)=3, L(5)=4,L(6)=4,L(7)=3

- O tamanho da maior subsequência crescente é

$$\max \{ L(1), L(2), \ldots, L(n) \}$$

- Temos a seguinte equação para L(j):
  [Tente para L(6) com A= (2, 3, 14, 5, 9, 8, 4)]

$$L(j) = \max\{1, \max_i \{ 1+L(i) \mid i < j \text{ e } a(i) < a(j) \}, \text{ para } j>1 \}$$
$$L(1) = 1$$

Maior subsequência crescente: encontrando o tamanho

```
Input: n, a_1,…,a_n ,

L(1)←1, pre(1)←0
For j=2 to n
  L(j)←1, pre(j)←0
  For i=1 to j-1 // faz L(i) <- max_i { 1+L(i) | i < j e a(j) > a(i) }
        If A(i) < A(j) and 1+L(i)>L(j) then
            L(j) ← 1+L(i)
            pre(j)←i     // atualiza predecessor
        End If
  End For
End For

MSC ← 0
For i=1 to n            // encontra maior L
  MSC ← max{ MSC, L(i) }
End For
```

[Fazer traço pra exemplo A = (4, 2, 3, 5) com pre(.)]

- pre(j): é utilizado para guardar o predecessor de j na maior subsequência crescente
- Complexidade $O(n^2)$

Encontrando a subsequencia (recursivamente)

Q: Algoritmo anterior calcula tamanho da maior subsequencia crescente. Como encontrar a subsequencia em si?

A: 1) Encontra L(j) com maior valor

2) Adiciona j a solução, segue pre(j), adicionando, etc.

```
Input: n, L(1),…,L(n),pre(1),…,pre(n), OPT

j←0
While L(j)<> OPT  //encontra L(j) com maior valor
   j++
End While
Find_Subsequence(j)

Proc Find_Subsequence(j)
   If j=0
     Return
   else
      Add j to the solution;
      Find_Subsequence(pre(j))
   End if
  Return
```

Complexidade O(n)

Maior subsequência crescente

Exercicio: Escreva a versão da programação dinâmica com memoização para resolver esse problema de maior subsequencia crescente

Exercise: Placing billboards

Placing billboards

Exercise: You need to decide where to put multiple advertisement on a highway of M kms.

- There are n possible places where you can place an advertisement given by x1, x2, .., xn in [0, M].
- Placing an advertisement at xi gives value ri.
- You cannot put two advertisements at distance < 5kms from each other.
- **Goal:** Find best set of places to put advertisement.

Ex: M=20, {x1,x2,x3,x4} = {6, 7, 12, 14}, and {r1,r2,r3,r4} = {5,6,5,1}.
One optimal solution is to put advertisement at x1 and x3

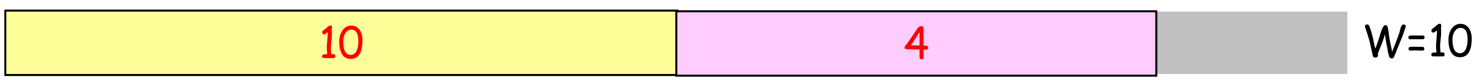
Solve this problem using dynamic programming

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

- Given n objects and a backpack of size W
- Item i has size $w_i > 0$ and has value $v_i > 0$.
- Sizes are integers.
- **Goal:** pick set of items that fit in the backpack and maximize total value.



Ex: { 3, 4 } has value 40.

W = 11

Item	Value	Size
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

.

Knapsack Problem: Greedy Attempt

Greedy: repeatedly add item with maximum ratio v_i / w_i.
Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal

.

W = 11

Item	Value	Size
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Dynamic Programming: False Start

Q: Sub-problems?

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }
- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing which other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with occupation limit w.

Q: What is a recursive expression for OPT(i, w)?

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using size limit w
- Case 2: OPT selects item i.
  - new size limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new size limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\;\; v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Dynamic programming. Fill up an n-by-W array to compute OPT(i,w)

Q: In which order should we fill this array?

A: Start with OPT(0, 0), then OPT(0, 1), OPT(0, 2)...; then OPT(1,0), OPT(1,2),...

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size! The input size is (log W + n)
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

Knapsack Problem

Exercise: Write down a pseudo-code to give what are the items in the optimal solution (the previous algorithm only gives the value of the optimal solution)

Exercise: A moving consulting company

A moving consulting company

Exercise: You have a small consulting company. Your clients are mostly in Rio and Sao Paulo.

- In each month it can run its business either from a Rio office or Sao Paulo office.
- In month i you have cost R_i if run from Rio, and S_i if run from Sao Paulo
- If you run the business from one city at month i and another at month i+1, then you need to spend a fixed cost M for moving costs.
- **Goal:** Given n months, decide where your office should be in every month to minimize total cost

Ex: $M = 10$, $\{R_1, R_2, R_3, R_4\} = \{1, 3, 20, 30\}$, $\{S_1, S_2, S_3, S_4\} = \{50, 20, 2, 4\}$
Optimal solution is [Rio, Rio, SP, SP], with cost 1+3+2+4+10=20

1. Show that the strategy of running the office from the city with smallest costs in each month does not minimize the total cost

2. Solve this problem using dynamic programming

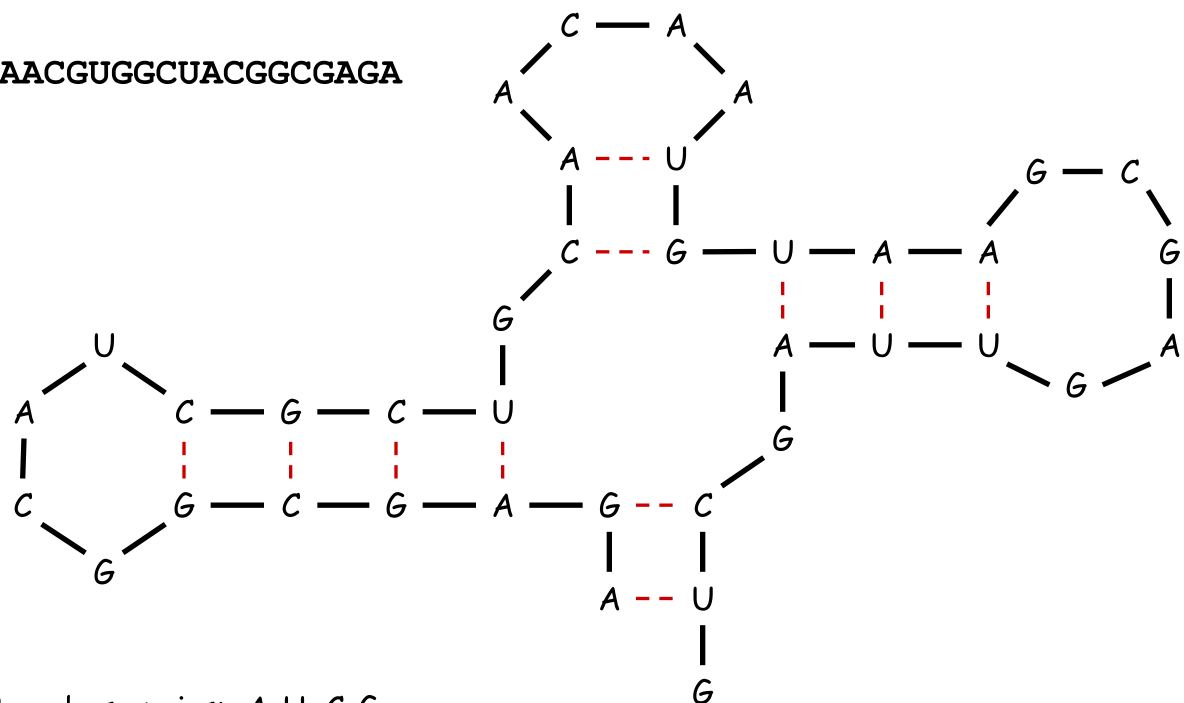
6.5 RNA Secondary Structure

RNA Secondary Structure

RNA. String B = $b_1 b_2 \ldots b_n$ over alphabet { A, C, G, U }.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**



complementary base pairs: A-U, C-G

RNA Secondary Structure

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

- [Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C.
- [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing.] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$.

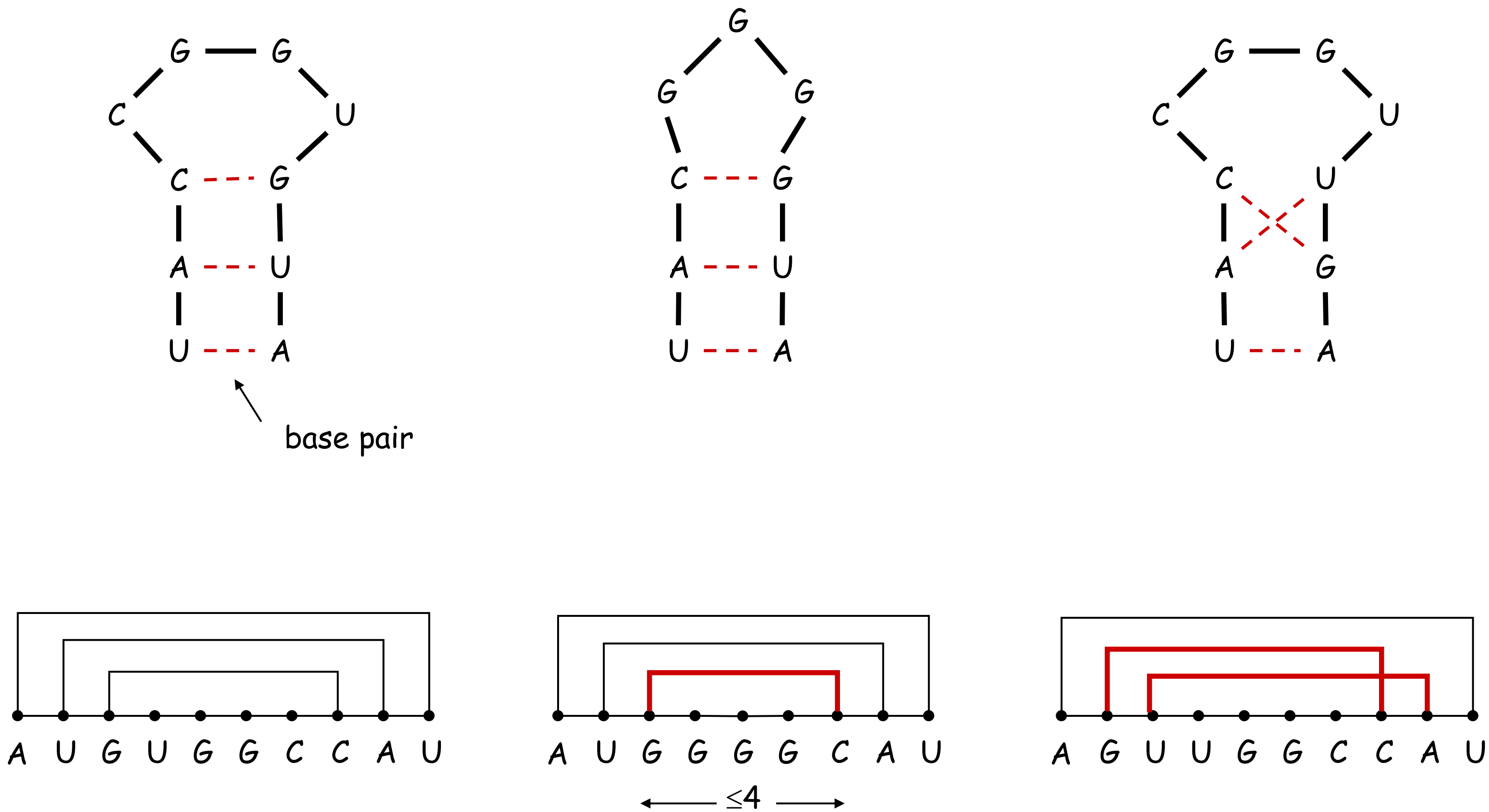
Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

approximate by number of base pairs

Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA Secondary Structure: Examples

Examples.



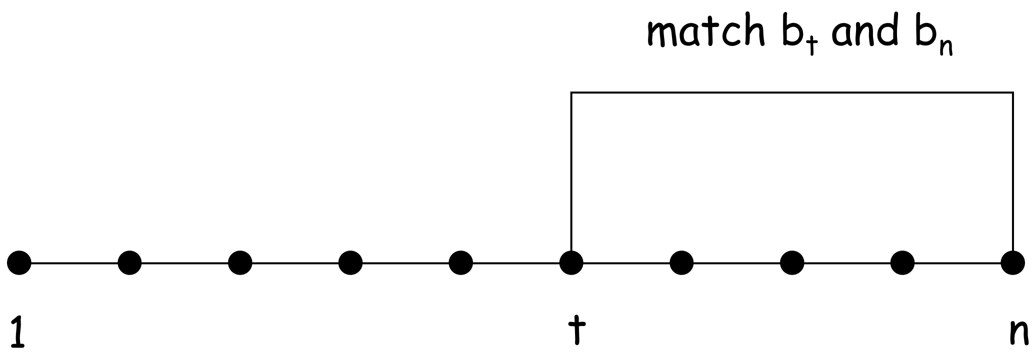
ok

sharp turn

crossing

RNA Secondary Structure: Subproblems

First attempt. OPT(j) = maximum number of base pairs in a secondary structure of the substring $b_1b_2 \ldots b_j$.



Difficulty. Results in two sub-problems.

- Finding secondary structure in: $b_1b_2 \ldots b_{t-1}$. ← OPT(t-1)
- Finding secondary structure in: $b_{t+1}b_{t+2} \ldots b_{n-1}$. ← need more sub-problems

Dynamic Programming Over Intervals

Notation. $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$.

- Case 1. If $i \geq j - 4$.
  - $OPT(i, j) = 0$ by no-sharp turns condition.

- Case 2. Base b_j is not involved in a pair.
  - $OPT(i, j) = OPT(i, j-1)$

- Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$.
  - non-crossing constraint decouples resulting sub-problems
  - $OPT(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$

take max over t such that $i \leq t < j-4$ and b_t and b_j are Watson-Crick complements

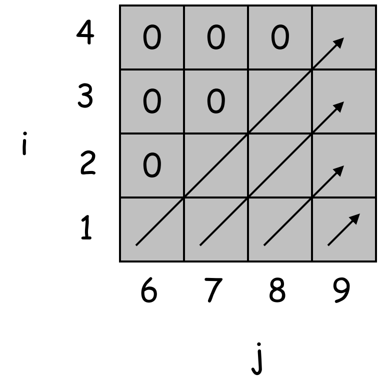
Remark. Same core idea in CKY algorithm to parse context-free grammars.

Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?
A. Do shortest intervals first.

```
RNA(b1,…,bn) {
   for k = 5, 6, …, n-1
      for i = 1, 2, …, n-k
         j = i + k
         Compute M[i, j]

   return M[1, n]
}
```

using recurrence



Running time. $O(n^3)$.

Dynamic Programming Summary

Recipe.

- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy

CKY parsing algorithm for context-free grammar has similar structure

Top-down vs. bottom-up: different people have different intuitions.

6.6 Sequence Alignment

String Similarity

How similar are two strings?

- `ocurrance`
- `occurrence`

Idea: Use gaps to align the strings, count #mismatches

(gaps = inserted/deleted character)

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

0 mismatches, 3 gaps

Edit Distance

Applications.

- Basis for Unix diff.
- Auto correction, spell checking
- Computational biology.

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

- Gap penalty δ; penalty α_{pq} for matching letter p with q (usually 0 if p=q)
- Cost: sum of gap and matching penalties.

Goal: Given two strings $X = x_1 \, x_2 \, . \, . \, . \, x_m$ and $Y = y_1 \, y_2 \, . \, . \, . \, y_n$ find alignment of minimum cost.

C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	T	A	C	A	T

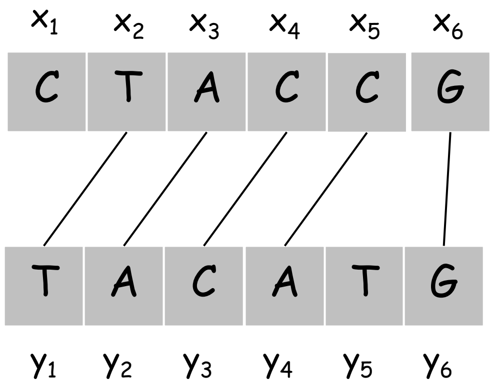
$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$

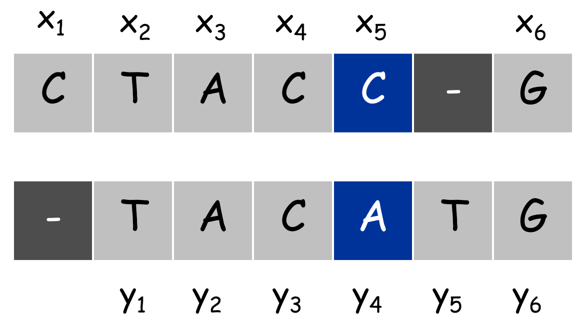
-	C	T	G	A	C	C	T	A	C	C	T
C	C	T	G	A	C	-	T	A	C	A	T

$2\delta + \alpha_{CA}$

Sequence Alignment

We can this as a matching problem:

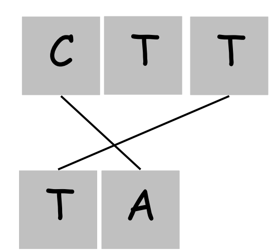




Cost of matching:

- Pay gap penalty δ for each unmatched letter
- Pay matching penalty α_{pq} for each matched pair

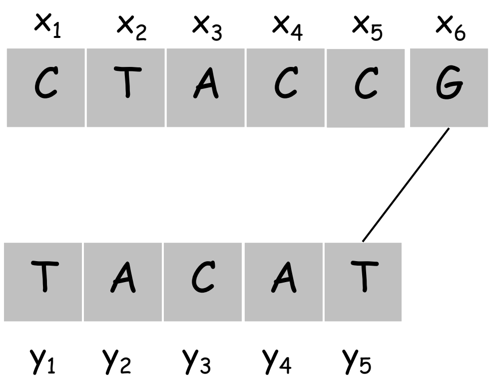
Crucial property: Valid matching does not have crossings (otherwise cannot represent using gaps)



Sequence Alignment: Optimal substructure

Opt. substr: OPT(i, j) = min cost of aligning strings $x_1\, x_2 \ldots x_i$ and $y_1\, y_2 \ldots y_j$.

Q: How can we write OPT(i,j) in a recursive way, in terms of smaller subproblems?



Option 1: solution matches last characters x_i and y_j

- Best to do is pay penalty α_{x_i, y_j} + best matching of rest $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_{j-1}$ (=OPT(i-1,j-1))

Option 2: does not match last characters.

Then one of them has to be unmatched, otherwise there is crossing

- Option 2a: leave x_i unmatched
  - Best is to pay gap for x_i + best matching of rest $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_j$ (=OPT(i-1, j))
- Option 2b: leave y_j unmatched
  - Best is to pay gap for y_j + best matching of rest $x_1 \ldots x_i$ and $y_1 \ldots y_{j-1}$ (=OPT(i, j-1))

Sequence Alignment: Optimal substructure

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
   for i = 0 to m
      M[i, 0] = iδ
   for j = 0 to n
      M[0, j] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                       δ + M[i-1, j],
                       δ + M[i, j-1])
   return M[m, n]
}
```

Analysis. $\Theta(mn)$ time and space.

English words or sentences: $m, n \leq 10$.

Computational biology: $m = n = 1.000.000$. 1 trilions ops OK, but 1 Tb array?

6.7 Sequence Alignment in Linear Space

Sequence Alignment: Value of OPT with Linear Space

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {

 for i = 0 to m
      CURRENT[i] = iδ
 end for

 for j = 1 to n
      LAST←CURRENT        % vector copy
      CURRENT[0]← jδ
      for i = 1 to m
         CURRENT[i] ← min(α[xi, yj] + LAST[i-1],
                          δ + LAST[i],
                          δ + CURRENT[i-1] )
 end for
return CURRENT[m]
```

- Two vectors of of m positions: LAST e CURRENT
- O(mn) time and O(m+n) space

Sequence Alignment: Value of OPT with Linear Space